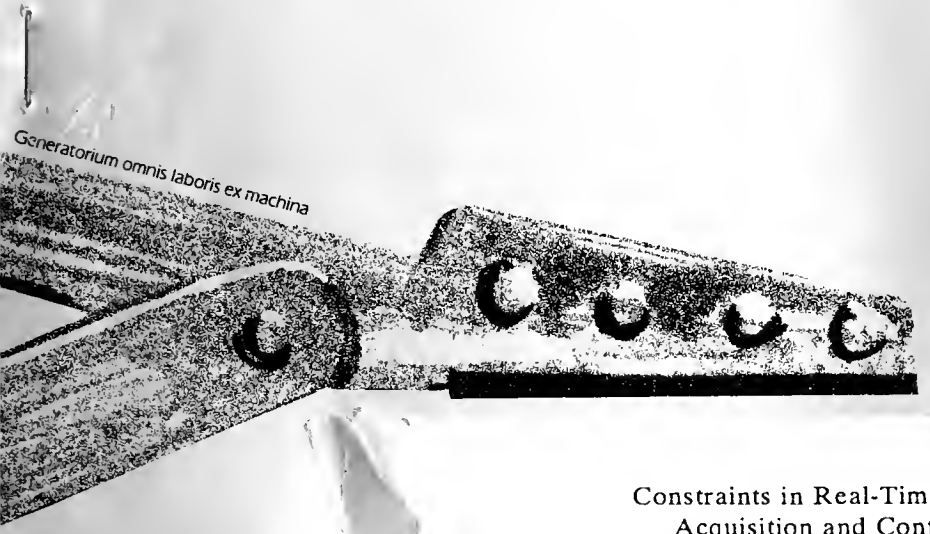
Robotics Research Technical Report

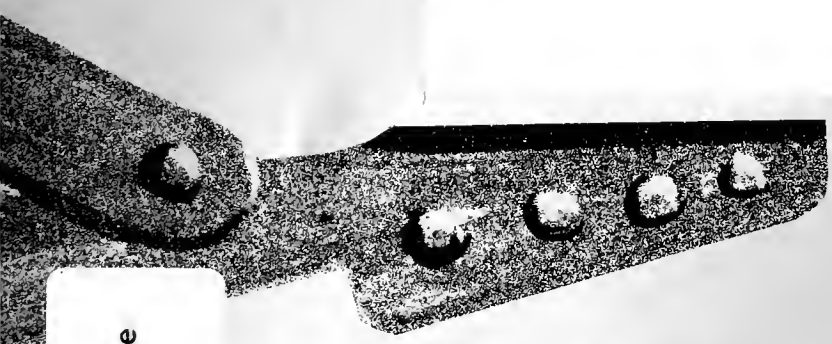


Constraints in Real-Time Data Acquisition and Control

by

Herbert J. Bernstein

Technical Report No. 195
Robotics Report No. 59
December, 1985



New York University
ant Institute of Mathematical Sciences

Computer Science Division
251 Mercer Street New York, N.Y. 10012

Constraints in Real-Time Data Acquisition and Control

by

Herbert J. Bernstein

Technical Report No. 195
Robotics Report No. 59
December, 1985

New York University
Dept. of Computer Science
Courant Institute of Mathematical Sciences
251 Mercer Street
New York, New York 10012

Work on this paper has been supported in part by the Department of Energy under contract DEACO276ERO3077-V, the National Science Foundation, under contract DCR-8511302, and the Office of Naval Research, under contract N00014-82-K-0381.

Constraints in Real-Time Data Acquisition and Control †

Herbert J. Bernstein

Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, New York 10012

Abstract

For control applications, computing speeds have increased much less dramatically over the past two decades than they have for numeric applications. Memory, cpu and bus speeds still limit control loops to significant fractions of a millisecond, and realistic robotic applications are contrained by such times. This lack of freedom in hardware timing then makes it difficult to use the software tools that have improved productivity in other areas. This paper reviews some of these constraints on real-time data acquisition and control for general robotic applications, and considers the prospects for their resolution.

Introduction

We consider the progress made in removing the constraints imposed by hardware and software in the design of real-time data acquisition and process control systems. Our thesis is that little has changed for most applications, other than the costs, but that the rapid decrease in cost makes possible an explosive extension of automation.

We will discuss

- The role of robotics in the factory;
- The meaning of *real-time* in data acquisition and control;
- How far we have progressed and are likely to progress in hardware;
- How far we have come in software;
- Solvable real-time problems; and

† Work on this paper was supported in part by the Department of Energy under contract DEACO276ERO3077-V, the National Science Foundation, under contract DCR-8511302, and the Office of Naval Research, under contract N00014-82-K-0381.

This talk will be presented at Automation '86, 9-12 March 1986, in Houston, Texas. A preliminary version was presented at the Symposium on Factory Automation and Robotics, 9-11 September 1985, in New York.

- Robot vision as an example of a real-time problem.

There is much to be done. Despite a significant increase in raw computing power for numeric problems, most of the techniques used do not have as great an effect on control problems. In perhaps a circular manner, control problems now handled by computers tend not to require much processing speed. However, as vision and AI techniques become an integral part of control loops, we can expect the computational demands of data acquisition and control to grow sharply.

Our context is that of the factory floor, which, until recently was dominated by people working within the limits of their sensory feedback and robots working *open loop*, *i.e.* in blind ignorance of the actual state of the workspace. This made the real-time demands on robot design rather minimal. However, sensory feedback has long been a part of many related systems of automata. Aircraft autopilots and instrument landing systems, building environmental control systems, automatic elevators and chemical plant process control systems are all examples of real-time closed loop servo-control systems. As factory automation and robotics mature, we can expect to see the lessons learned in these other areas applied on the factory floor, and techniques developed for robotics applied elsewhere.

Robotics in the Factory

Robots in the factory are part of the trend towards *flexible manufacturing systems*, (see [8]), in which machine tools, material handling systems and control computers have been brought together in an integrated manner to make the factory as a whole a smoothly running machine. In this concept, the factory consists of manufacturing cells into which raw materials and parts are brought to be fashioned into finished products or parts for use in other cells.

Notice that this model of the factory can be viewed as a network, which can be analyzed with tools from graph theory. The factory faces problems similar to those seen in rail yards, telephone systems, and data communications. The materials transport system provides the communications medium and the cells are the communicating nodes. This should be still more the case with flexible robot based cells displaying highly variable service times, than with more deterministic assembly lines. We would then have a generalization of a Jackson network with blocking [6] in which average delays and queue lengths could be modeled. For the purposes of this paper, however, let us ignore the important questions of the effect of blocking, and consider only the network flows aspect. In 1956, Ford and Fulkerson [5] showed that

> The maximum flow between any two arbitrary nodes in any graph cannot exceed the capacity of the minimum cut separating the two nodes. (The *Max Flow Min Cut* Theorem),

i.e. the flow is limited by the capacity of the narrowest bottleneck. This fundamental result applies to the factory as a whole, to the materials

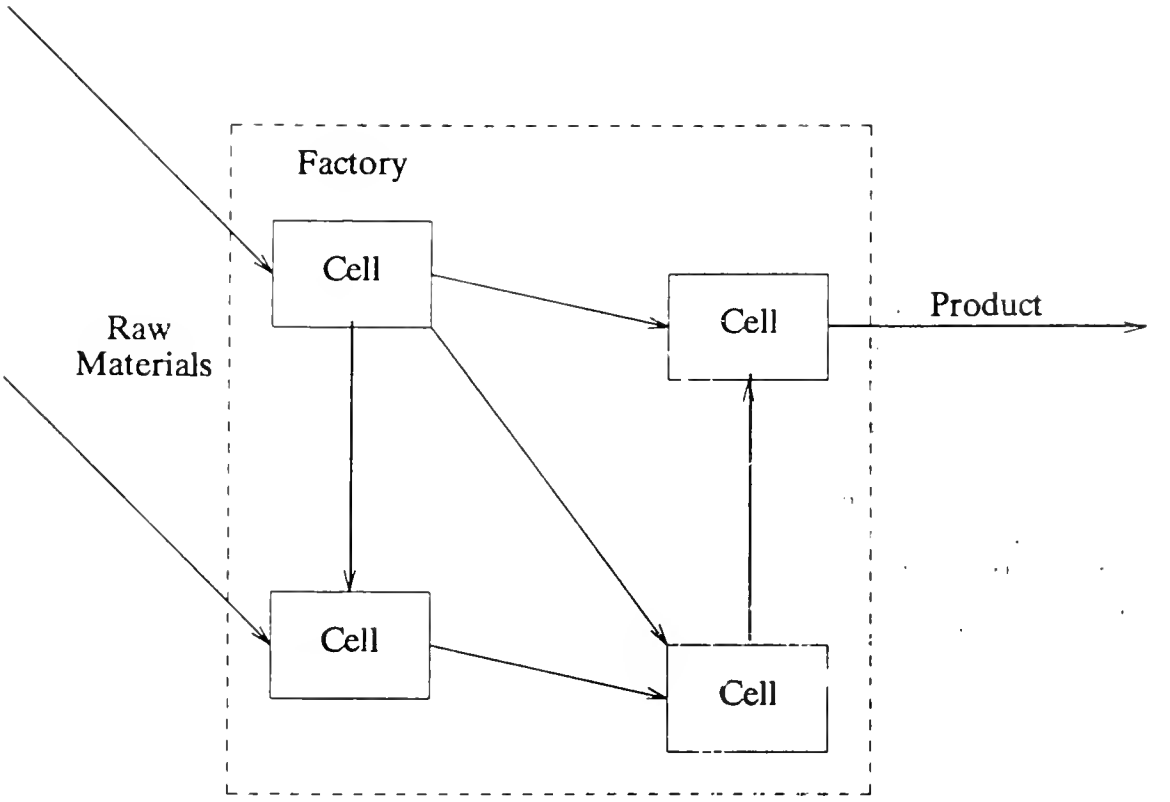


Figure 1. Flexible Manufacturing System model of the factory, consisting of manufacturing cells, materials transport system, control and MIS system.

transport system, and to the computers used. As a consequence, we can expect heartbreakingly little improvement in overall factory performance until the last bottleneck is removed. Indeed, steps which may lead to this long term desirable result may be singularly cost ineffective in the short term.

Let us take the long view, and assume the further automation of manufacturing cells is desirable. To this end, consider the control system of a cell. For each of the control systems involved, the cell provides a *workspace*, in which various *effectors* are brought to bear according to some planned actions intended to implement a *task definition*. The control system shares the common failing of people: it can never deal with the true workspace, only with some internal *model workspace*. Thus the expected results of its planned actions may not agree with the actual results. Parts may be missing from the input feed. Tools may wear. A jig may shift. A hole may be misplaced by a few thousandths of an inch.

A solution is to have *sensors* in the workspace which provide information on actual results to compare to the expected results. The difference between actual and expected results can then be used to correct the planned actions. This structure is shown in Figure 2. The results of the comparison need not,

of course, be used only to correct the already planned actions. They could also be used to revise the model and the plans themselves, moving us into more adaptive control.

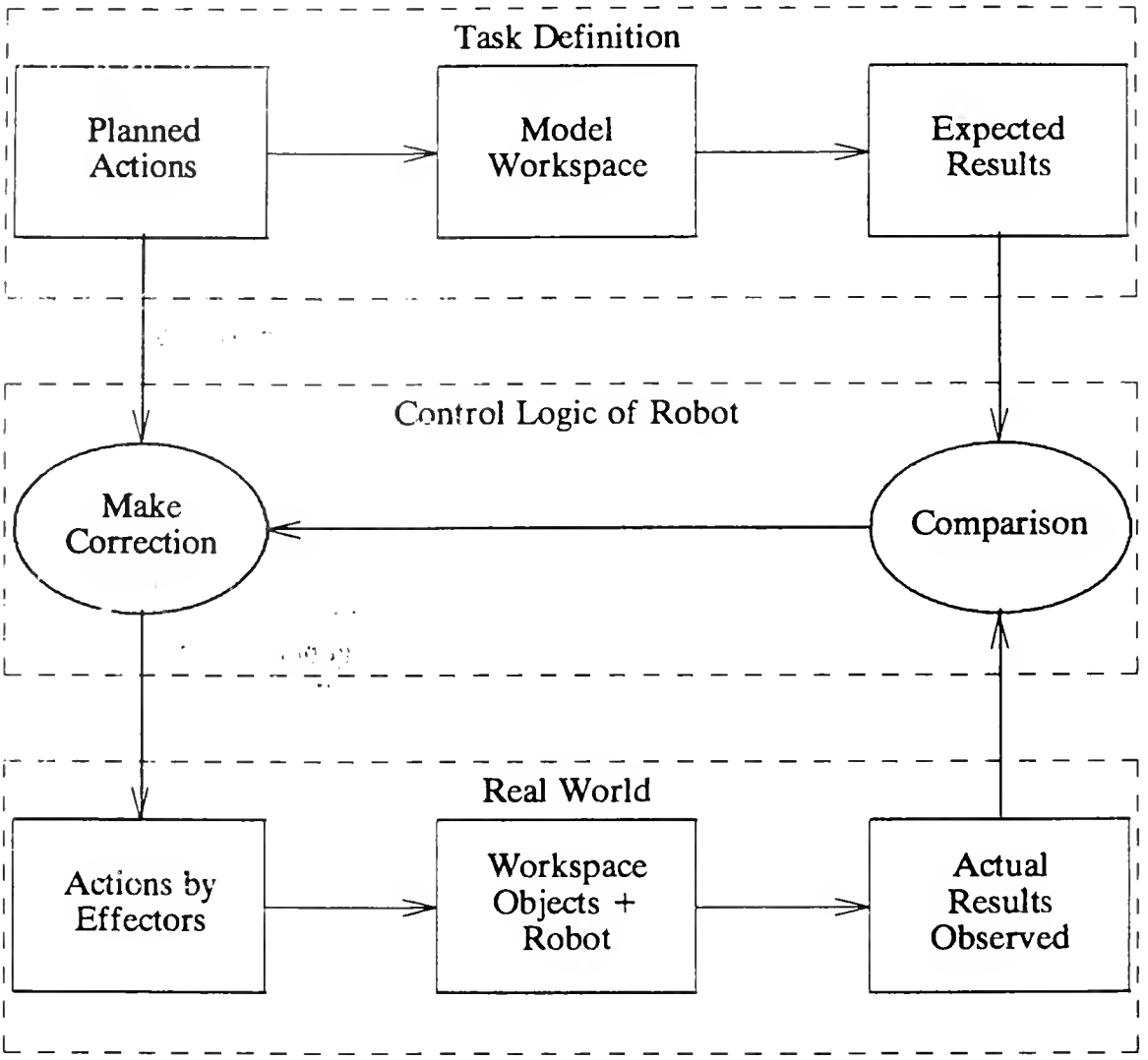


Figure 2. Servo control model of a robot work cell. Expected results are compared with observed results to compute corrections to actions by effectors in the workspace.

The timing of a control cycle is shown in Figure 3. We can minimize overall delays by making the model computation time no longer than the total time needed to make corrections, have the effectors act and sense the results. We should also make the planning time for a cycle no longer than the total time needed to do the previously mentioned steps and perform the comparison. These optimizations are rarely trivial, since there are usually complex transformations among the parameter spaces involved. The effectors may be working with, say, a robot's revolute joint parameters,

while the model was formulated in the ordinary Cartesian coordinates of the laboratory floor. This creates hidden transformation functions in the arrows of Figure 3.

A change in one of these hidden functions can make the time spent in one path of Figure 3 longer while making the time spent in another path shorter, effectively moving time in the diagram. Let f_{MC} be the transformation from the model space of parameters to that used in the comparison. Let f_{SC} be the transformation from the sensor space of parameters to that used in the comparison, *i.e.*

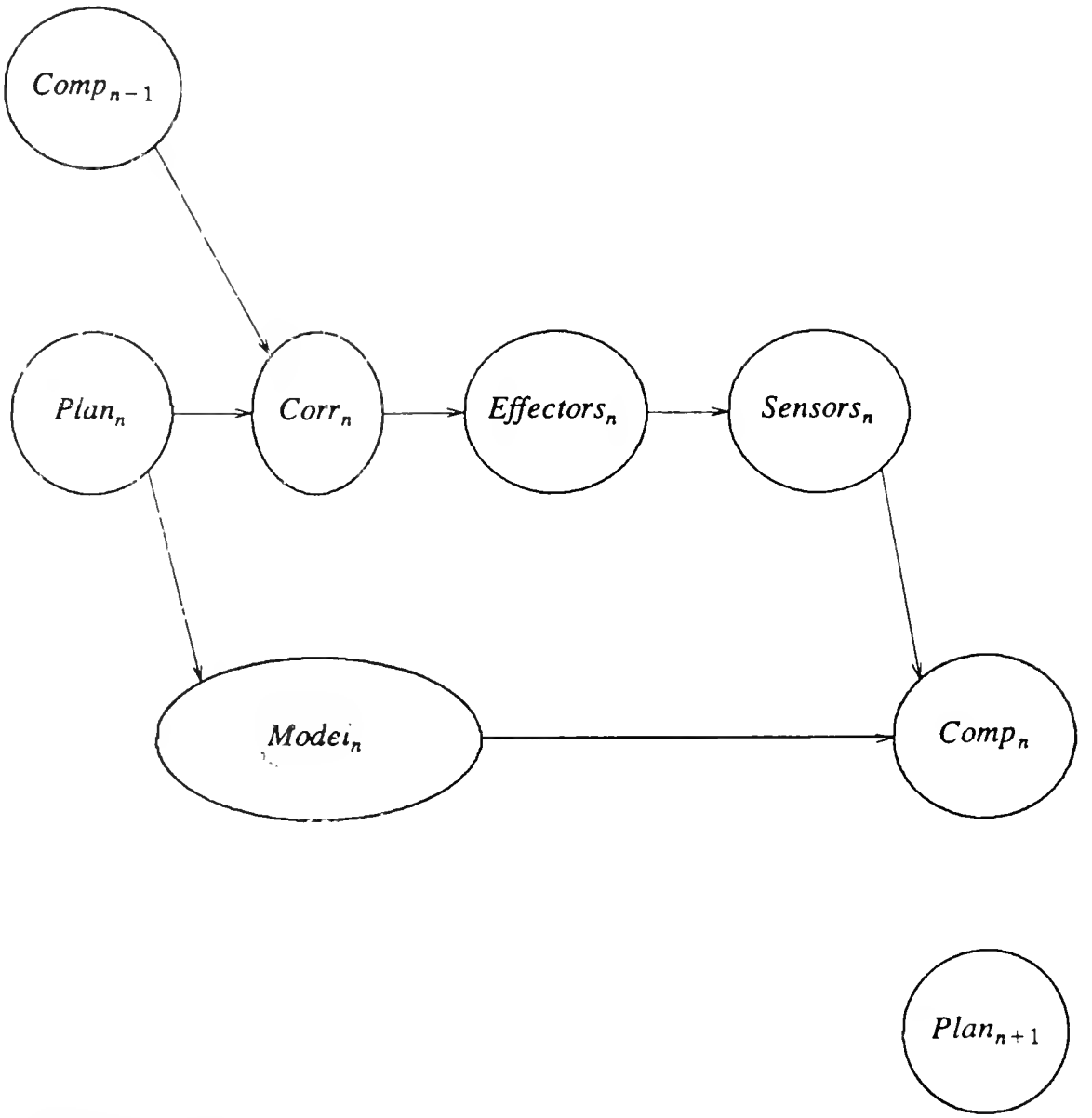
$$f_{MC}:\{Model\} \rightarrow \{Comparison\}$$
$$f_{SC}:\{Sensors\} \rightarrow \{Comparison\}$$

If our model time is too long, we can shift some of that time into the sensor logic by changing the transformations and the comparison logic, so that comparisons are done in the model space of parameters, *i.e.*

$$Comparison' = f_{MC}^{-1}(Comparison)$$
$$f'_{MC} = f_{MC}^{-1} f_{MC} = I,$$
$$f'_{SC} = f_{MC}^{-1} f_{MC}.$$

If we are lucky, and are dealing with simple linear transformations, it may be possible to find transformations which save time on both paths by simultaneously diagonalizing matrices. In general, however, the transformations are not so nice, and we have a complex parallel program optimization to perform.

Even when the transformations are trivial, the time can still be tight. Simply picking up data and putting it down can take a significant fraction of control loop time. It takes time for the effectors to act, still more time for the sensors to report actual results, time to make comparisons, and time to compute the necessary corrections. Thus by the time the corrections are ready to be applied to the planned actions, they are really appropriate to the past rather than the present. If the total of all the times involved is small compared to the rate at which the situation changes in the workspace, the corrections will still be approximately correct, and may be applied in time to make actual results and expected results come closer together. If the time lags involved are significant, it becomes necessary to anticipate the situation at the time when the corrections will be applied, rather than using past parameters (*feedforward* rather than *feedback*), in order to be able to maintain control. This technique makes it possible to work with a control system that responds on the time scales of the workspace. However, if the control system responds even more slowly than that, there is little hope of maintaining any meaningful control. A *real-time* data acquisition and control system is one in which the total of all the time lags involved is sufficiently small to allow us to maintain control.



Desirable Timing:

$$T_{Model} \leq T_{Correction} + T_{Effectors} + T_{Sensors}$$
$$T_{Plan} \leq T_{Correction} + T_{Effectors} + T_{Sensors} + T_{Comparison}$$

Figure 3. Servo control timing. The times along alternate paths should be balanced to minimize waiting time.

Real Time Data Acquisition and Control

Mok [9] says

> "Computer Systems which must continuously observe critical timing constraints are said to function in a *hard-real-time* environment."

That is our context. In a factory setting, real-time is defined by the production flow of the industry in question. For a chemical plant, reactions may take days or small fractions of seconds. In the first case a control system could include the postal system and still function, while in the second no human being could respond in time to avoid loss of a production run, or even loss of life. In painting an auto body, stopping a spray arm for half a second every few seconds could produce an uneven coat, while in welding that same auto body, pauses of minutes between welds do no more than slow the line a bit. We need some way to establish a realistic time scale for real-time.

Human beings provide one benchmark. We handle visual data with a response time on the order of 250 milliseconds, and tactile data on a time scale of about 60 milliseconds [7]. It is reasonable to expect similar responses from a supposedly real-time computer based control system working with factory cells derived from human parameters. We may, however, have to insist on still tighter control.

If we have a robot arm moving at 50 meters per second, and expect to control its trajectory on a grain of 1 centimeter steps, we will have to react as much as 5000 times per second, bring our control loop time down to 200 microseconds. We can certainly think of tighter constraints in realistic situations, but, as we shall see, even these mild time assumptions will press the limits of the capabilities of current hardware and software. Most control computers are actually rather slow, and seem likely to remain so.

Every few years someone discovers that computers and electronics are getting faster, cheaper and more reliable, and announces to the world that the millennium in cost effective real-time data acquisition and process control is upon us. Then some cynic rises and points out that programmers make mistakes, hardware breaks and costs money and that *real* real-time problems are still an order of magnitude too difficult to solve. Both the optimist and the cynic are out of sync with reality. One is trying to apply solutions that are not yet available to problems that cannot wait, while the other is ignoring real progress in the art. A balance must be struck. If we are to do useful work with rapidly advancing technology we must make a realistic assessment of the constraints imposed by software and hardware available at reasonable cost in a useful time frame.

Indeed, we can think of real-time control system design as a servo control system itself. The time lags through the design and fabrication process are at best months and probably years. Our best control of the design process comes when we successfully anticipate the available technology on such a time scale. If we are too conservative, we waste valuable production time. If we are too optimistic, our system may never work. One reasonable rule of thumb is to never design on the basis of an

undelivered paper product, but otherwise use products from reliable vendors early in their production life cycle.

This being said, we should note that for most applications, we are still working with control cycles of times similar to those of two decades ago. Why?

How Far Have We Come?

If we look at early relay computers able to do at most a few hundred simple operations per second, and compare them to the latest supercomputers able to do more than 1,000,000,000 complex floating point operations per second, we would seem to have gained a factor of more than a million in computing speed in the past few decades. I regret to say that, from the point of view of real-time control, a good part of this gain is illusory, and much of the real advances were made before 1960. Since then, significant speed increases have been difficult to obtain.

To understand these points, first we must decide what it is we are comparing. There are many time scales to consider in computing. Raw logic elements are only capable of switching at certain rates, given by rise-times and gate-delays. Logic elements are put together to form more complex circuits, such as flip-flops, which can change state at even slower rates. Many layers of such circuits are then needed to make major portions of computers, limiting the computer's *clock rate* to still slower rates. Each instruction may then in turn require some substantial number of clock cycles to complete. Most users of computers want to know the rate at which they can execute instructions, but the very techniques needed to make computers run faster, *e.g.* parallel execution, pipelining, memory caching and interleave, make it difficult to define a meaningful instruction execution rate. For example, is a machine consisting of 100 processors each capable of executing a million instructions per second to be considered faster or slower than a machine with a single processor capable of executing 100 million instructions per second?

In real-time work, events are happening at a rate comparable to the time it takes to queue tasks for individual processors, so unless we have an application in which the allocation of processors can be predetermined to avoid the cost of processor assignment in the middle of servicing an event, the effective speed of a control computer is that of a single processor. Such speeds are strongly tied to raw logic speeds.

In 1959, Erich Bloch [2] presented the design of the Stretch Computer. If that design had been fully realized, it would have reached an instruction rate of 2 Mips, based on 4 way-interleaved 2 microsecond memory, and a 200 nanosecond memory bus cycle. The design had a high degree of parallelism which could have done a floating addition in 1 microsecond.

At the same time Jan A. Rajchman [11] analyzed the application of tunnel diodes, phase-locked oscillators and microwave circuitry borrowed from radar work, to produce logic circuits. He produced circuits using flip-flop clock rates in excess of 100 MHz, memories with cycle times of less than

100 nanoseconds, and showed that flip-flop clock rates in excess of 300 MHz were feasible, based on 10 GHz diodes with rise times of 2 to 3 nanoseconds. He suggested that "microencapsulated improved diodes promise to provide thousand megacycle [1GHz] cycle rates with power supplied at 30KMC or higher."

Using more ordinary components, it was possible to build 30MHz flip-flops well before 1960. Smith [12] describes *toggles*, i.e. flip-flops in that range in 1959.

By 1968 commercially available logic speeds had advanced to 500 MHz flip-flops, 1 nanosecond gate delays and rise times [15]. In August 1985, typical small computers reached speeds of 1 to 3 Mips [1] while mainframes could deliver 15 Mips per cpu, with two to four processors per mainframe [13], and supercomputers were in the range of 100 to 250 Mips per processor, again with two or more processors per mainframe. Gallium arsenide logic, which is just becoming available, is able to reach flip-flop clock rates of 3GHz. Thus, in terms of logic speeds, comparing commercially available logic of 1960 to that of 1985, we have obtained a speed improvement by a factor of 100. This is also true in comparing the scalar instruction execution rates of the best computers of 1960 to those of 1985. While this is not a factor of a million, it is at least respectable. It is a bit worrisome that the best experimental logic speeds have not improved as much, say only by a factor of three, but at least there is still some room for further improvement in raw computer logic speeds.

However, when we compare ordinary computers, rather than supercomputers, there is much less reason to be pleased. The IBM 7090 of 1959 [14] was effectively a 1/5 Mips machine, having a 2.18 microsecond memory cycle time, and a reasonable number of two cycle instructions. The norm for most ordinary computers today is in the range of 1/2 to 2 Mips, an improvement by only a factor of 10, not what we would expect looking at the high end. The problem appears to be memory and bus speeds.

It lowers the cost of a computer to make use of memories which are very dense. While one could make a very fast memory out of low to medium density logic, and one does do so for internal machine registers, most memory is made out of very dense chips, with cycle times of 100 to 250 nanoseconds for dynamic RAMs and 30 to 70 nanoseconds for lower density static RAMs, packaged in large blocks with common control logic. That control logic itself adds some tens of nanoseconds to a memory cycle. Larger machines then bring their timing back into line by having many memories interleaved. Smaller machines settle for fewer memories and are then hard pressed to recover the lost speed.

Even so, one would expect to be able to reach 5 to 10 Mips even in a small machine, and some of them are in that range. Most, however, do not make it because of the inherent limitations of busses.

One can make a computer more modular and maintainable by standardizing the interconnections between major subsystems. One approach is to define a common set of wires to which many subsystems will be

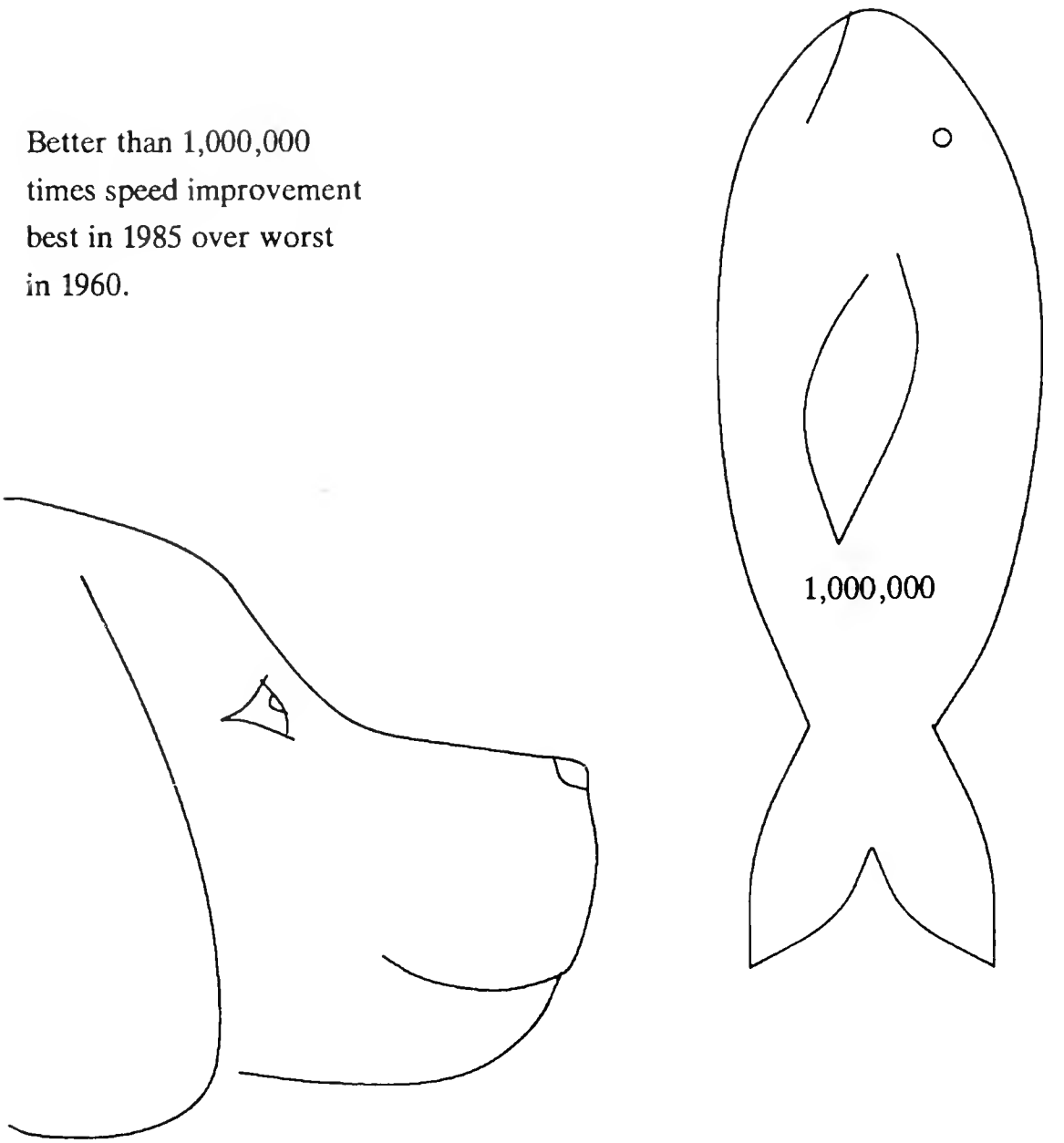


Figure 4. The improvement from the worst computer speeds of 1960 to the best computer speeds of 1985 is better than 1,0000,000 times. However, this is a bit of a fish story.

connected. When two of the subsystems need to communicate, they somehow seize the common *bus*, use it, and then relinquish it for other subsystems to use. Usually, one would expect the CPU to be master in a steady stream of reads and writes to and from memory.

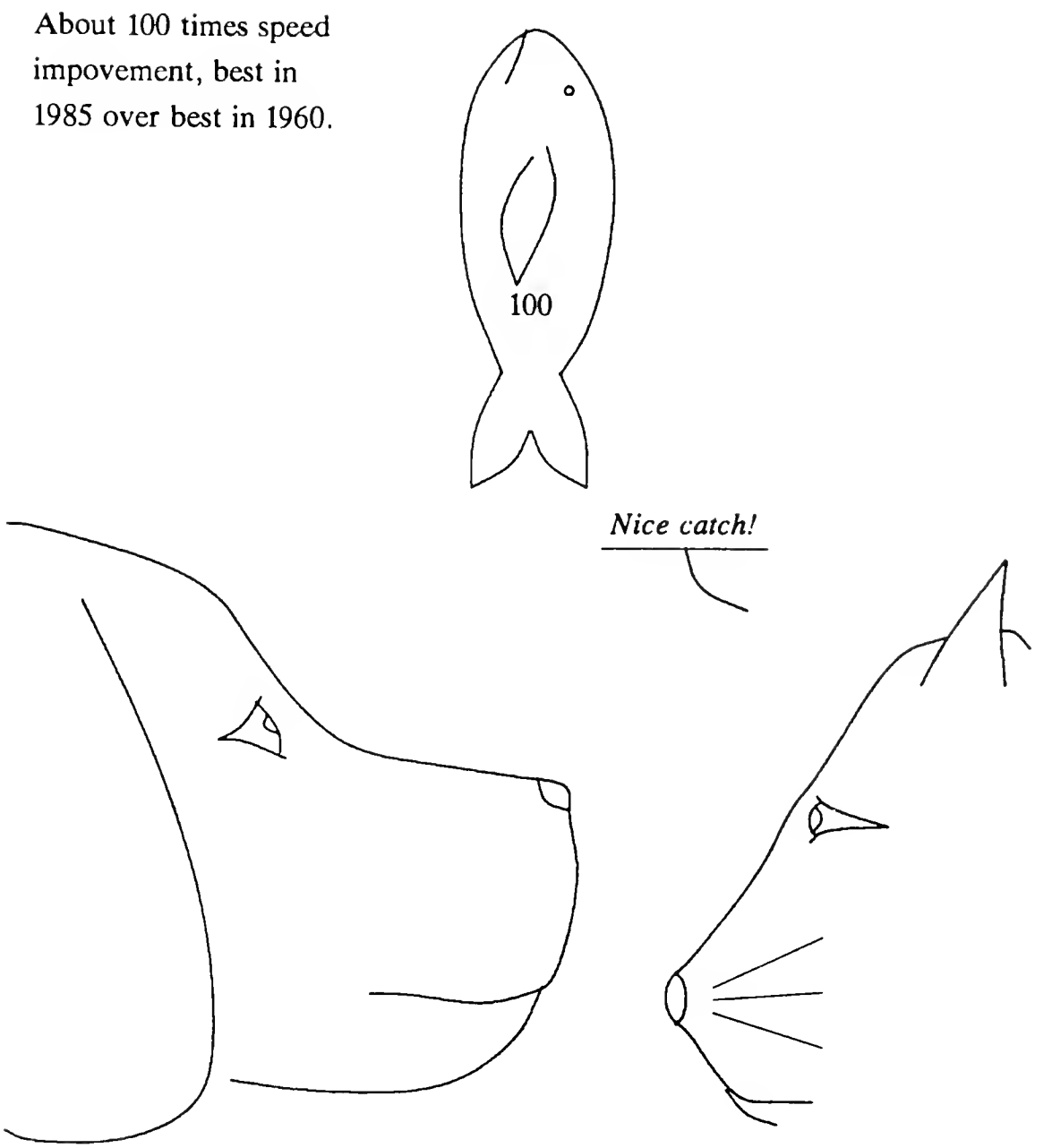


Figure 5. The improvement from the best computer speeds of 1960 (the 2 MIPS Stretch design) to the best computer speeds of 1985 (200 MIPS supercomputer scalar speeds) is about 100 times. The fish is a smaller but still respectable.

In order to use a bus consisting of multiple parallel wires reliably, one has to contend with two physical parameters: settling time and deskew time, shown in Figure 7. When a signal is imposed on a bus wire, it take some

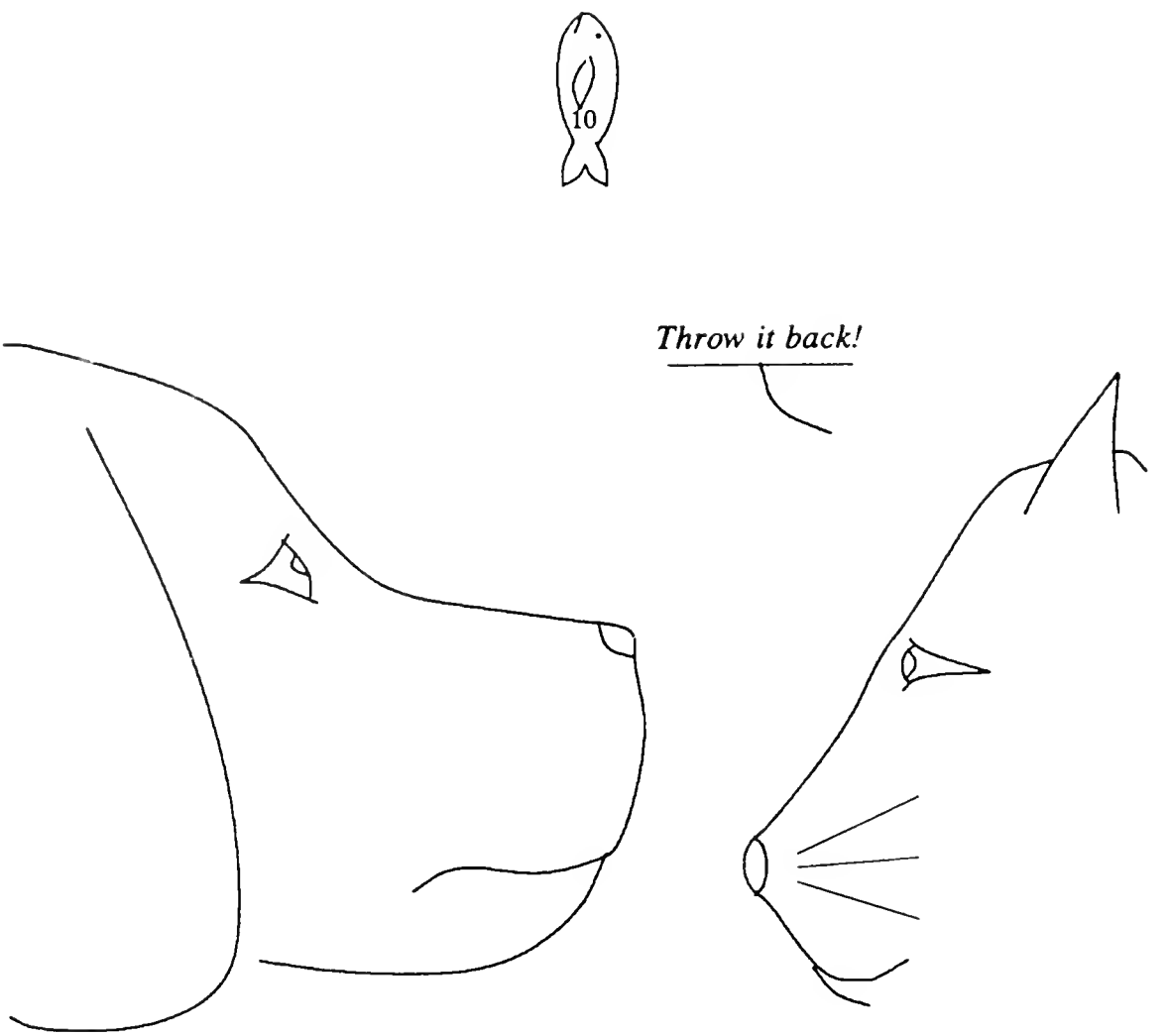


Figure 6. The improvement from the average computer speeds of 1960 (better than 1/5 MIPS on 7090) to average computer speeds of 1985 (1 - 2 MIPS VAXes and the like) is only about 10 times. This is especially true for control computers.

time to charge the capacitance of the bus. Typical busses have an effective source impedance of about 100 ohms. A common bus, by its nature has significant length and several taps, contributing to capacitances in the range

of 50 pf or more, causing settling times in the tens of nanoseconds. When multiple bus lines are used, we cannot be certain that all of them will have the same settling times, nor that all of them will be driven by gates of identical characteristics, nor even that all of them will be driven through the same number of gate delays. Thus, when working with multiple parallel bus lines, it is possible for signals on one line to be skewed in time relative to signals on another. Before we can believe any signal on a bus, we must wait until it has settled, and, if that signal is composed of multiple parallel signals, we must wait still longer to resolve the question of skew.

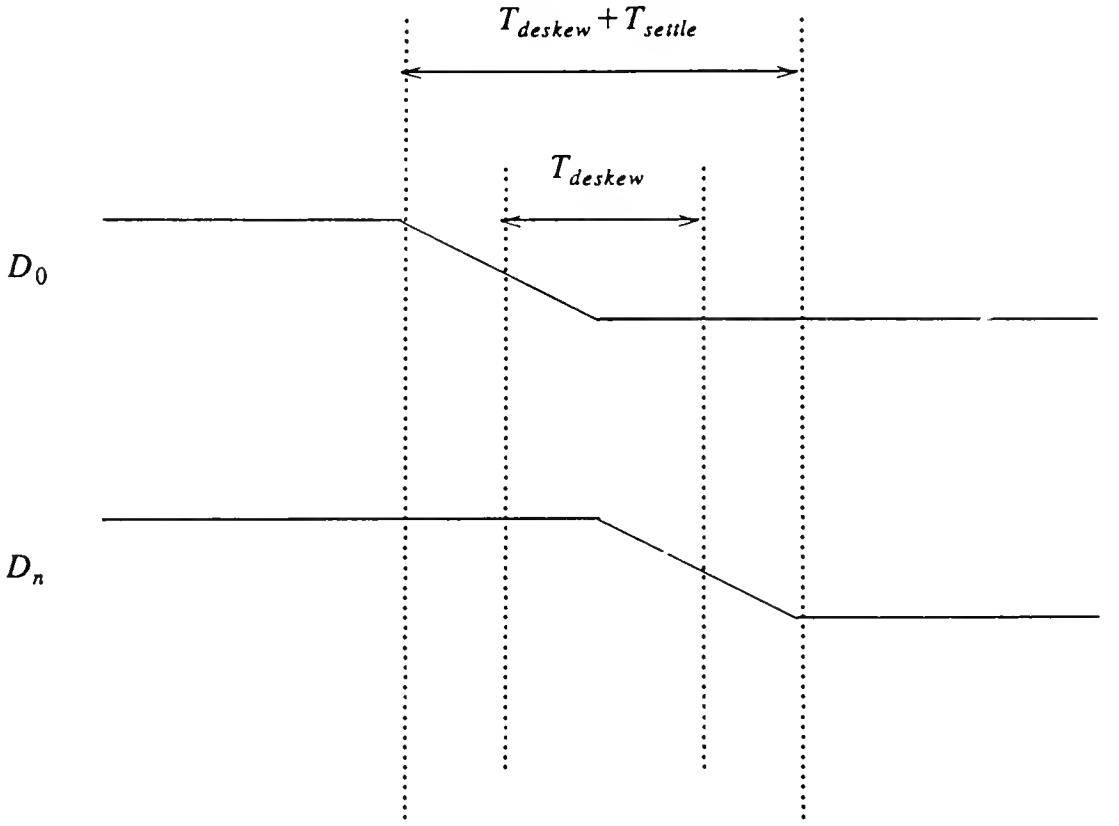


Figure 7. Deskew and settling time for bus data signals.

Given a memory of, say, 50 nanosecond access time, we must add to that time a settling and deskew time for address assertion, a settling time for an address strobe, a settling time for data assertion, and a settling time for a data strobe. Before we can use the bus again, we must wait long enough for address and data from the previous cycle to be removed.

Some overlap among these times is possible. The settling time of the strobes can be overlapped with the deskew times. The time to clear the address from a previous cycle can be invisible if the address lines are not the same as the data lines. Even so, we can expect to face two to three settling and deskew delay times in every bus cycle, typically adding more than 200 nanoseonds to the memory access time. Using a memory cache can and does

help, but as long as the cpu must also deal with a bus on some regular basis, bus memory cycle speeds limit the overall processing speed.

When dealing with such times, it is not surprising that instruction rates have not gotten far beyond 1 Mips in ordinary computers. Clearly, much can be gained by moving away from parallel busses as the standard interconnect, and using either faster serial busses, or point-to-point interconnection networks.

Hardware Constraints

As we have noted above, the dominant hardware constraint in control design is cpu and memory cycle time. Typical control computers have instruction cycle times on the order of a microsecond. In a control cycle, the processor must obtain the current state of all relevant workspace sensors, obtain the expected state of those same sensors, compare the values and update the data to be applied to the workspace effectors. In the simplest case, one processor is responsible for a single sensor/effector pair, say a joint position encoder and the joint motor. In this case, all it need do is a digital or analogue input of a single value, subtract that from some known target position, scale to make a properly damped correction, and add to the next target value. Each step could take 10 microseconds, and we would still have done the task in less than 50 microseconds, leaving time for communication with some host which is establishing the target values for the control processor to use.

When we move up to multiple coupled parameters, however, we find the tasks more demanding. Suppose we have a 6-joint robot with a geometry which creates couplings among the joints, and wish to control the motion of the robot on the basis of force sensors. We have 6 effector parameters to output, 6 position sensor and 6 force sensor inputs to read. The correction process may well involve applying a 6 by 12 matrix to the entire 12 component sensor vector. Thus a control cycle may now require 72 multiplies, 66 adds, 12 external inputs and 6 external outputs. If each multiply requires 5 microseconds, each add only 1 microsecond, and the external operations 10 microseconds each, we face a loop time of at least 706 microseconds. We could not keep up with our hypothetical 50 meters per second motion in 1 centimeter steps.

Because of such limitations, control cycles in the .1 to 1 millisecond range are still the norm. For most mechanical systems, the natural resonant frequencies are even slower, so little is lost. Indeed, it is hard to make effectors which respond in less than 50 microseconds, making any attempt to provide new information at a faster rate pointless. However, as we move to control algorithms involving multiple coupled sensors and effectors, the computational load rises. If our force sensors are used at multiple points, say on several finger tips, and we attempt coupled control of, say, 5 4-jointed fingers, we may well be faced with a 20 by 44 control matrix. pushing our loop time to over 5 milliseconds, assuming some other processor worries about all the bookkeeping of computing the matrices.

On top of this already significant load, we have to add code to detect the fact that we have entirely left the control regime in question, or that we have hit some sort of physical limits. Assuming no additional sensors to be handled for such cases, we have still more matrices to apply to vectors, and some number of comparisons to do. These steps can consume as many cycles as the control matrix application. bringing the minimal control loop up to 10s of milliseconds.

Even this is not all we must contend with. If we attempt to share one processor among multiple control tasks, we must allow for time to perform a context switch. The more powerful the processor, the more state information we are likely to have to save. A simple one accumulator CPU need only store the accumulator, a few status bits, and a program counter to switch context. A cpu with 16 general registers may well have to store all 16 and fetch a full new set on every context switch, imposing a price of many tens of microseconds for each such switch. In such an environment, almost any processor looks slow.

Software Constraints

Recently, Michel Parent and Claude Laurgeau [10] wrote

> "the microprocessors used in robotics ... are usually programmed in *machine code (assembler)* for reasons of performance, and to allow efficient management of real time tasks."

This is an indication of the difficulty of real-time software development. In software, the biggest problem is still one of avoiding programmer errors. There have been many efforts to convert programming from an art to a science. Top-down programming, data-flow analysis, and software engineering have contributed to a better understanding of the mechanisms of software design for complex projects. We have Pascal, C and Ada to help us avoid the need for unmaintainable assembly language code. In control applications, however, where every cycle counts, the temptation is still there to write the guts of the code in assembly language.

Even the minimal convenience of an operating system is often denied us. In the *hard* real-time regime in which we are working, we must be certain of our response times. A typical multitasking operating system can have internal queues with delays of many hundred of microseconds, an unacceptable situation for control loops on the same scale.

Ada [3] tries to correct this bias towards real time programming in assembly language by providing a complete programming environment for computers embedded in larger systems imposing real-time constraints. It will be interesting to see if it makes significant inroads into an area long resistant to modernization.

For the moment, an effective compromise seems to be the use of a hierarchy of control computers. The truly real-time tasks are given to small slave computers (*e.g.* joint servos) which can be programmed on a low level, say in assembler or in C with a little assembler, and then locked up. These lowest level control computers are then treated as more timing tolerant

devices by a host running with the less precise timing of an operating system. An equivalent approach is to use a large enough, fast enough machine to be able to unconditionally reserve all the cycles required for the real time task as the highest priority of the system, with a normal operating system left to fend for itself on whatever cycles the real-time task does not take away. In either case, assuming that the hard real-time routines have been given their necessary resources, the higher levels of software can be written in higher level languages without much problem.

Thus both hardware and software are making slow progress in performance. The real progress in control systems has been in costs. In the 1950s, a control application which could justify the expenditure of only a few hundred dollars had to be done with electromechanical cam timers, a few relays and a bit of analogue feedback. Computer control was available, but only at prices three to four orders of magnitude higher. By the 1960s, the same applications could use purely electronic systems at similar costs, but programmability was mainly by such things as diode pinboards. Computer control was still in the several tens of thousands of dollars range. By the 1970s computers started to become sufficiently inexpensive to be considered as part of low-cost control systems. First minis dropped below the $10,000 mark and then micros became available. By the 1980s, one could buy fast reliable 4-bit micros able to do everything a cam timer could do at much lower cost. We now see costs down to the point where a processor for every actuator and sensor is feasible.

Solvable Real Time Problems

Where are we likely to go from here? There are two directions to consider: How can we improve our available tools; and how can we make best use of the tools available.

We can improve the design of computers for real-time work. We can move away from parallel busses to serial busses and point-to-point interconnects. We can design processors with many register sets to make the costs of context switching minimal. We can integrate low startup matrix operations into small machines. However, judging by the past 25 years, we had best not expect too rapid an advance in raw processing speeds, and devote our energies to solvable problems.

We can divide real-time problems into three categories:

- Those for which no full solution is available,
- Those for which a solution is known, but the costs are too high, and
- Those for which a cost effective solution is available.

Our objective is to recognize when a problem takes a step forward on this list.

An example of a problem for which no full solution is now available, is the simple human task of driving a delivery truck in traffic. The complex of object recognition, motion planning and servo feedback problems are simply beyond the state of the art. Note that in driving a truck, we do not have the

luxury of stopping the clock while we cope with a slow bit of software. Control must be maintained at all times. We also do not have the luxury of full information about our environment prior to system operation. Despite the best of traffic reports, an accident may block a road, or a construction crew may force a detour. If we ever do make a control system for this problem, it will have to be self-training and have a reserve of raw processing speed sufficient to get it out of worst case situations. Progress in sensor systems, control systems, artificial intelligence, and high performance computer design is needed before we can make a major dent in such a problem.

In the middle ground of problems with known, but expensive solutions, are robotic vision systems for use in factories. Given a large enough supercomputer and enough vision sensors in enough locations, we can recognize the limited class of objects likely to be found in a factory workspace in real-time. If we lag a bit, we can always make the production process pause. If we find a surprise in the field, such as a human hand, we can either steer clear, or blow the whistle on the intruder and shut down. The problems here are ones of improving known approaches to data acquisition and data reduction to increase the range of suitable applications.

Vision as an Example.

Coiffet [4, pp 198-203] summarizes the progress in vision system response times through 1981. The times ranged upwards from 300 milliseconds, which is quite respectable. However, such speeds are obtained by imposing drastic restrictions on the class of objects to be recognized. We could broaden that class by using very large computers, because vision problems do contain significant number crunching aspects. In general however, we would want the vision processing to be of a cost comparable to that of the mechanical systems involved, requiring us to take some care in the design.

Despite long processing times, vision systems are making significant inroads into factories. In a conveyor belt system, we can compensate for vision processing time by moving the vision system back along the conveyor until the time of travel is a least as long as the processing time. This matching of pipelines is not an effective solution for robots.

Consider the measures needed to make cost-effective vision systems for robots when we cannot wait for, or afford, a sufficiently fast computer. If the vision system is to be used as a servo-sensor, the total image processing time must be on the time-scale of the mechanical systems involved, say no worse than 1/10th of a second. If we assume that we will cut our hardware costs by using a pipeline of conventional image processing stages handling ordinary television images, our time constraint implies a pipeline of at most 3 stages, each working in a frame time. This is restrictive. We can improve the timing by giving up some resolution, say by working with field stages instead of frame stages, or by making later pipeline stages work faster than frame times. In any case, we need a partitioning of the object recognition

problem into fixed-time stages, e.g. component recognition, object boundary parameterization, object recognition. When each of these stages has a cost-effective hardware/software implementation which run in small fixed time *independent of scene complexity* we will be able to consider making widespread use of such systems on general purpose robots in factories. Until then, we will still need people to help robots see.

References

[1] "32-bit microsystem hits 2.7 MIPS," *Electronic Products*, pp. 23-24, August 1, 1985.

[2] Erich Bloch, "The Engineering Design of the Stretch Computer," in *Proceedings of the Eastern Joint Computer Conference, December 1-3, 1959, Boston Massachusetts*, pp. 48-58, The National Joint Computer Conference, 1959.

[3] Grady Booch, *Software Engineering*, The Benjamin/Cummings Publishing Company, Menlo Park, California, 1983, 504 pp.

[4] Philippe Coiffet, *Interaction with the Environment*, Robot Technology, 2, Kogan Page, London, England, 1983, 240 pp.

[5] L. R. Ford, Jr. and D. R. Fulkerson, "Maximal flow through a network," *Canadian Journal of Mathematics*, vol. 8, pp. 399-404, University of Toronto Press, Toronto, Ontario, Canada, 1956.

[6] B. Gopinath, editor, *Computer Communications*, Proceedings of Symposia in Applied Mathematics, Vol. 31, American Mathematical Society, Providence, Rhode Island, 1984.

[7] G. Hirzinger, "Direct Digital Robot Control Using a Force-Torque Sensor," in *Real Time Digital Control Applications, Proceedings of the IFAC/IFIP Symposium, Guadalajara, Mexico, 17-19 January 1983*, ed. A. Alonso-Conchero, pp. 243-255, Pergamon Press Ltd., London, England, 1984.

[8] Tom Klahorst, "The History of FMS Control," *Commline*, September/October 1981. As reprinted in *Flexible Manufacturing Systems*, Society of Manufacturing Engineers, Dearborn Michigan, 1984, pp 3-6.

[9] Aloysius K. Mok, "The Design of Real-Time Programming Systems Based on Process Models," in *Proceedings Real-Time Systems Symposium, December 4-6, 1984, Hyatt Regency Hotel, Austin Texas*, pp. 5-17, IEEE Computer Society Press, Silver Spring, Maryland, 1984.

[10] Michel Parent and Claude Laurgeau, *Logic and Programming*, Robot Technology, 5, Kogan Page, London, England, 1984, 190 pp.

[11] Jan A. Rajchman, "Solid-State Microwave High Speed Computers," in *Proceedings of the Eastern Joint Computer Conference, December 1-3, 1959, Boston, Massachusetts*, pp. 38-47, The National Joint Computer Conference, 1959.

[12] Charles V. L. Smith, *Electronic Digital Computers,* McGraw-Hill Book Company, Inc., New York, New York, 1959, 443 pp.

[13] Norman Weizer, "Sierra: Where Will it Lead?," *Datamation*, vol. 31, pp. 84-92, Technical Publishing, New York, New York, May 15, 1985.

[14] *Reference Manual IBM 7090 Data Processing System, Form A22-6528-4,* International Business Machines, September 1962, 156 pp.

[15] *Motorola MECL Integrated Circuits,* Motorola Inc., Pheonix, Arizona, 1978, 255 pp.

N.Y.U. Courant Institute of
Mathematical Sciences
251 Mercer St.
New York, N. Y. 10012